Wait-Free Clock Synchronization[*]

Shlomi Dolev[†] Jennifer L. Welch[†]

May 31, 2001

Abstract

Multiprocessor computer systems are becoming increasingly important as vehicles for solving computationally expensive problems. Synchronization among the processors is achieved with a variety of clock configurations. A new notion of fault-tolerance for clock synchronization algorithms is defined, tailored to the requirements and failure patterns of shared memory multiprocessors. Algorithms in this class can tolerate any number of *napping* processors, where a napping processor can fail by repeatedly ceasing operation for an arbitrary time interval and then resuming operation without necessarily recognizing that a fault has occurred. These algorithms guarantee that, for some fixed k, once a processor P has been working correctly for at least k time, then as long as P continues to work correctly, (1) P does not adjust its clock, and (2) P's clock agrees with the clock of every other processor that has also been working correctly for at least k time. Because a working processor must synchronize in a fixed amount of time regardless of the actions of the other processors, these algorithms are called *wait-free*. Another useful type of fault-tolerance is called *self-stabilization*: starting with an arbitrary state of the system, a self-stabilizing algorithm eventually reaches a point after which it correctly performs its task.

Two wait-free clock synchronization algorithms are presented for a model with global clock pulses. The first one is self-stabilizing; the second one is not but it converges more quickly than the first one. The self-stabilizing algorithm requires each processor's communication register contents to be a part of the processor's state. This last requirement is proven necessary. A wait-free clock synchronization algorithm is also presented for a model with local clock pulses. This algorithm is not self-stabilizing.

Key words: Distributed Computing, Algorithms, Wait-Free, Self-Stabilization, Clock-Synchronization.

[†]Department of Computer Science, Texas A&M University, College Station, TX 77843. Email: `shlomi@cs.tamu.edu` and `welch@cs.tamu.edu`. Fax: 409-847-8578.

[*]This work was supported by NSF Presidential Young Investigator Award CCR-9396098 and Texas A&M University Engineering Excellence funds. A preliminary version of this work was presented in the 12th ACM Symposium on Principles of Distributed Computing, August 1993.

1 Introduction

Multiprocessor computers are being designed with ever-increasing numbers of processors. These multiprocessors can be used to solve problems that demand high computation power, such as grand challenge computing problems, which previously were not efficiently solvable. However, in order to take full advantage of multiprocessors, it is vital that they be made fault-tolerant. Fault-tolerance is necessary in order to provide even the same level of availability that is provided by uniprocessors, since the probability of a crash in a multiprocessor system increases with the number of processors. Clever fault-tolerance schemes may also be able to provide a higher level of availability, by continuing ongoing computations even if a large number of processors fail.

A central issue for any multiprocessor system is the synchronization among processors. The common synchronization component used in multiprocessors is a *clock*. There are several ways to implement a clock in multiprocessor systems: (1) provide a common clock that is connected to all the processors in the system, (2) provide a common clock pulse that reaches every processor in the system and stimulates individual clocks, (3) provide every processor with an individual clock pulse that stimulates its individual clock. The less centralized the clock unit is, the more reliable the system can be, e.g., a system with multiple individual clocks may be able to tolerate a wrong behavior of one individual clock while a system with a central clock unit cannot. In the sequel we consider distributed clock synchronization algorithms for the case in which a common clock pulse reaches every processor or every processor has an individual clock pulse.

We present a new view of fault-tolerant clock synchronization, inspired by the architecture and failure-patterns of shared-memory multiprocessors (cf., e.g., [KS91, KP+92]). We are interested in clock synchronization algorithms that are highly resilient to failures. In particular, we want them to tolerate any number of processor failures and for nonfaulty processors' clocks to be unaffected by the failures. It is also important for processors that have ceased being faulty to be able to rejoin the system and become synchronized. More precisely, we require an algorithm to guarantee that, for some fixed k, once a processor P has been working correctly for at least k pulses, then as long as P continues to work correctly, (1) P does not adjust its clock, and (2) P's clock agrees with the clock of every other processor that has also been working correctly for at least k pulses.

The clock synchronization problem has been extensively studied in the presence of arbitrary, or Byzantine, faults (e.g., [Ma83, LM85, MS85, DHS86, ST87, WL88]). This fault model is so strong that no algorithm can work unless more than two-thirds of the processors are nonfaulty [DHS86]. Because of its high cost and the fact that it is often claimed to be too pessimistic, weaker fault models have been studied. A weaker fault model called authenticated Byzantine allows an algorithm that can tolerate any number of faulty processors [HSSD84]. However, in that algorithm faulty processors can influence the clocks of nonfaulty processors (namely, by speeding them up). Furthermore, in that failure model reintegration of repaired processors is only possible if less than half the processors are faulty.

In light of the above we choose to consider a more restricted type of faults, which we call "napping." A *napping* failure causes a processor to stop operation and then resume without necessarily recognizing that a failure has occurred. When the requirements above are stated for napping failures they capture the spirit of *wait-freedom.* In a wait-free system, each processor must solve the problem of interest in a finite (or bounded) number of its own steps, regardless of the speed of all the other processors. (Cf., e.g., [La86b, ALS90].) Therefore we call an algorithm that solves the problem described above in the presence of napping failures a *wait-free clock synchronization algorithm.*

One of our algorithms is also *self-stabilizing.* An algorithm is called self-stabilizing if it is resilient to transient faults in the sense that, when started in an arbitrary system state, if no further faults occur then the processors converge to a consistent global state and can solve the task. (Cf., e.g., [Di74, La86a, DIM90, AKY90, DIM91, AV91].) Self-stabilizing clock synchronization algorithms have been proposed [GH90, ADG91], but none of them is wait-free.

Self-stabilization and wait-freedom are incomparable conditions: a self-stabilizing algorithm works regardless of the state in which it is started, but requires that none of the processors fail subsequently, while a wait-free algorithm works regardless of whether some processors stop but relies on a correct initial state. Our work is among the first to address self-stabilization in combination with other failures. Independently, Gopal and Perry [GP93] have presented a self-stabilizing consensus algorithm that withstands omission faults in a message passing environment.

Most of the literature on clock synchronization concentrates on synchronization by exchanging messages. In contrast, in this paper we are interested in clock synchronization for shared memory multiprocessor systems. Towards this end we define several settings for clock synchronization algorithms in the context of shared memory systems. Our system model extends that of [GH90] and [ADG91] by allowing more relaxed clock primitives and the combination of failure types. We believe that the models we present allow us to address some of the inherent algorithmic issues for clock synchronization in shared memory multiprocessor systems. In addition, the models are abstractions of existing and future VLSI technology where a clock pulse is generated for synchronizing operations (e.g., [Ul84, Hw93]).

In this paper, we present several wait-free clock synchronization algorithms for different system settings. Except where noted, the clock values are integers that can grow without bound. In all cases, the n processors communicate through n shared variables, one variable associated with each processor. The first and second algorithms assume a global clock pulse; in this model (called the *in-phase* model in the sequel), at each pulse, every (nonfaulty) processor first reads one other shared variable and then writes its own. The first and second algorithms guarantee synchronization with $k = O(n^3)$ and $k = O(n^2)$ respectively. The $O(n^3)$ algorithm is self-stabilizing while the $O(n^2)$ algorithm is not.

The self-stabilizing algorithm requires the content of the communication register associated with a processor to be a part of the processor's state. We present an impossibility result that illustrates the necessity of this last requirement. Even under this restriction, the existence of an algorithm that can tolerate any initial state and up to $n-1$ faulty processors is somewhat surprising.

The third algorithm assumes that every processor owns an independent clock pulse; in this model (called the *out-of-phase* model in the sequel), at each (local) clock pulse a processor may read a shared variable and write its own shared variable. However, unlike the case in which a global clock pulse exists, both read and write actions are invoked simultaneously. Thus, the value written by a processor at a pulse is defined solely by the processor's state prior to the pulse occurrence. The third algorithm achieves $k = O(n^2)$; it is not self-stabilizing.

The remainder of the paper is organized as follows. In Section 2 we formalize the assumptions and requirements for a wait-free clock synchronization algorithm. Towards the presentation of our algorithms we present in Section 3 two impossibly results. The impossibility results give a better understanding of the difficulties that an algorithm has to cope with. Sections 4 and 5 contain the in-phase algorithms and the out-of-phase algorithm, respectively. We conclude in Section 6.

2 Problem Statement and Assumptions

In this section we define the requirements for a wait-free clock synchronization algorithm. Towards that end, we first describe our system models.

The system consists of n identical processors and n shared variables. A processor P is modeled as a (possibly infinite) state machine. Each shared variable is "owned" by exactly one processor; that processor is the only processor that may write to the variable, while all the processors may read from it. (A single-writer shared variable is a natural choice for achieving synchronization since a multi-writer shared variable might require some synchronization to begin with in order to guarantee atomic writes.)

A *configuration* of a system is a tuple containing a local state for each processor and a value for each shared variable. For self-stabilizing wait-free clock synchronization algorithms we prove that the shared variable value has to be part of the processor state. Thus, for such a system a configuration can be defined simply as a tuple of the processors' local states.

We consider two system models. In the *in-phase* model, there is a global pulse that triggers every non-faulty processor to take a step. A step of a processor consists of the processor reading one of the shared variables, performing some local computation which may change its local state, and then writing its own shared variable. The state of the processor just before the step occurs indicates which shared variable is to be read.

In the *out-of-phase* model, each processor has its own local pulse that triggers it to take a step. A step of a processor consists of the processor reading one of the shared variables and writing to its own shared variable; however the value written to its own variable cannot depend on the value just read. A processor can also change its local state during a step. The state of the processor just before the step occurs indicates which variable will be read and which value will be written.

We now define an execution for in-phase systems. A *pulse* is a (possibly empty) list of processor names, indicating which processors take a step at this pulse. An *execution* is a finite or infinite sequence $c_0 \pi_1 c_1 \ldots$ of alternating configurations and pulses, starting (and ending, if finite) with a configuration. (Pulse π_i will be referred to as pulse i and configuration c_i will be referred to as configuration i.) Pulses are consecutively labeled $j, j+1, j+2, j+3$, etc., for some j. The label of a pulse is the real time when it occurs, in an appropriate unit. Each configuration c_i (except the first) must correctly reflect the steps of all the processors in pulse π_i, based on configuration c_{i-1}. All the processors that are active at pulse π_i base their steps on the values of the variables and states in c_{i-1}, and then change state and write variables in unison to produce c_i.

If a processor is not included in a pulse, then it is said to *nap* (or not work) at that pulse. In an actual system, this could happen if a clock pulse does not reach a processor P either because P does not recognize its occurrence (since it is crashed or because of a transient bug) or due to a fault in the connection to P.

We now define an execution for out-of-phase systems. A *pulse* is a processor name, indicating the processor taking a step now. An *execution* is an alternating sequence of configurations and pulses as before. Pulses are consecutively labeled with increasing real numbers such that the difference between any two consecutive steps of the same processor is at least one time unit. The consistency constraints on configurations are the same.

For a given execution E we define a *sub-execution* of E to be an infix (i.e., a prefix of a suffix) of E that begins (and ends, if it is finite) with a configuration. Note that a sub-execution is also an execution.

If two consecutive steps of a processor occur more than one time unit apart, then the processor is said to *nap* (or not work) in the interval between the steps.

The following definitions are made with respect to a fixed execution E of either an in-phase or out-of-phase system. The *time* of configuration i is the label on pulse i (i.e., the real time of the pulse just before this configuration). The time of c_0 is one time unit less than the time of c_1. For configuration c, processor P, and component x of P's shared variable, $P.x(c)$ denotes the value of component x of P's shared variable in configuration c. For any real number (time) T, $P.x(T)$ is defined to be $P.x(c_i)$, where c_i is the latest configuration in E whose time is less than or equal to T.

The next main definition ("work") captures how long a process has been working; it requires an auxiliary definition ("last_step"). Let $\mathit{last\_step}(P, E, T)$ be the last time processor P executes a step before T during E. Define $\mathit{work}(P, E, T)$ to be zero if $(T - \mathit{last\_step}(P, E, T)) > 1$, otherwise $\mathit{work}(P, E, T)$ is the maximal l such that for every integer $m \leq l$, P executes a step at time $\mathit{last\_step}(P, E, T) - m$. Intuitively, $\mathit{work}(P, E, T)$ is the number of consecutive pulses P executed in E before T without napping.

In either in-phase or out-of-phase systems, an *initialized* execution is an execution whose first configuration is an initial configuration—a configuration with particular values as specified by the program.

We now define the wait-free clock synchronization problem. Each processor P's shared variable has a component $P.clock$. A system is running a *wait-free clock synchronization algorithm with convergence time* k, where k is a positive integer, if every initialized execution E of the system satisfies the following two conditions.

Adjustment: For all times T and all processors P, if $work(P, E, T) > k$, then $P.clock(T) = P.clock(T-1) + 1$.

We say that a processor P *adjusts* its clock during a pulse if P does not nap in this pulse and P does not increase its clock by 1 during the pulse. Intuitively, the adjustment requirement states that there exists a time after which a working processor does not adjust its clock.

Agreement: For all times T and all processors P and Q, if $work(P, E, T) \geq k$ and $work(Q, E, T) \geq k$, then
$P.clock(T) = Q.clock(T)$ for in-phase,
$|P.clock(T) - Q.clock(T)| \leq 1$ for out-of-phase.

A wait-free clock synchronization algorithm is *self-stabilizing* if it works correctly in *every* execution, starting with any configuration, not just in initialized executions.

2.1 Discussion of the Model

One may view both the in-phase and out-of-phase systems as extensions of the well known PRAM model (see e.g., [KR90, Le92]). The in-phase system is essentially a PRAM in which processors can experience faults, both transient and napping. There are synchronous multiprocessor systems that use shared memory or message passing communication to which our in-phase result could be applied (e.g., [Hw93] pp. 457). The out-of-phase case can be viewed as a more loosely coupled system in which processors do not share a common clock pulse. Since the assumptions made on the out-of-phase model are less restrictive, it abstracts an even wider range of existing architectures that use shared memory for communication [Hw93].

It is clear that, in addition to the fault tolerance aspects the choice made for implementing a clock in multiprocessor systems is influence by hardware considerations. Next we discuss the hardware implementation issues of such extensions of the PRAM model. The hardware designer of a multiprocessor system might have to cope with some of the following questions.

- Is the implementation of a global pulse much less difficult than the implementation of a reliable counter (central clock unit) whose value is increased at each pulse and sent out with the pulse?
- Can the synchronization of the clocks be implemented by a "reset" wire whenever the system is initialized or inconsistency of clocks is detected?

- Is it possible to implement a global pulse that reaches all the processors simultaneously and triggers operations at the same time (given the existing interrupt mechanism)?
- When every processor has a local clock is it possible to ensure that the interval between two pulses is exactly the same on all correct processors?

Unlike the fault tolerance aspect for which it is clear that as more the clock implementation is distributed is better the answers for the above questions may depend on the particular setting of the multiprocessor system in hand. Next we try to qualitatively answer the above questions.

- In some cases the implementation of a reliable counter which is connected to every processor might require more than a single wire (instead, a wire per each bit of the counter) to be connected to each processor. This in turn will exacerbate the issues of power used to drive the signals and the chip space wasted [MC80].
- A common reset signal might imply that any inconsistency detected is handled globally by resetting the clocks of all the processor to some predefined value — in some case this approach could be too drastic. For instance, when a single processor fails and then resumes operation, we might not like the rest of the processors to reset their clock values. Moreover, since every processor may initiate a reset request, some synchronization mechanism might be needed to avoid too many signals taking place at the same time using the same reset wire.
- The implementation of global clock pulse might not be possible because of the propagation delay and the nature of the interrupt handler. However, it is possible to ensure that during a pulse every active processor first reads and only after all the read operations are executed do the write operations take place. This is sufficient for our in-phase algorithms.
- It might not be possible to devise clocks that have the exact same rate. Moreover, even if such clocks are implemented, it is hard to ensure that processors will take actions at the same rate (especially when the pulse triggers an interrupt). However, for our out-of-phase algorithm it is only important to maintain the same relative order between the processors for long enough. This is possible when the individual pulses are spread away from each other to begin with. This can be achieved with high probability by the use of a random number generator (in the beginning and) following every time a processor discovers that it napped. The processor will use the random number generator to determine the time it needs to wait until its hardware clock produces the first clock pulse. In addition, because of clock drift, the pulses of two active clocks might become close to each other or even change their relative order. Our out-of-phase algorithm will consider such a change as a nap of the slower processor(s).

3 Impossibilities

In this section we present some of the difficulties that must be surmounted in devising an algorithm.

3.1 Hidden Values

Suppose at each clock pulse, each processor could read the entire shared memory, not just a single shared variable. In this case there is a simple self-stabilizing algorithm with convergence time $k = 1$: namely, in each step the processor reads all the *clock* variables in the system, finds the maximal value max and assigns $max + 1$ to its *clock*.

The obvious adaptation of this simple algorithm to the in-phase model, in which each processor can only read a single shared variable at each pulse, is to have each processor read all the clock variables in the system in a succession of steps, and then set its clock to one more than the maximum value observed. We now show that this approach does not work.

In more detail, we assume that every processor reads the clocks of the other processors in a cyclic order, one after the other. Whenever a processor P reads $Q.clock$ and finds that $Q.clock > P.clock$ then P assigns $P.clock := Q.clock + 1$. Otherwise, when $Q.clock \leq P.clock$, P assigns $P.clock := P.clock + 1$. We call this the "first shot" algorithm.

We now present a specific execution of the above algorithm for which there is no k that satisfies the adjustment requirement. The difficulty intuitively is that the maximal clock value can remain hidden from nonfaulty processors arbitrarily long. This execution, denoted by E, is for a system with four processors. Part of E is depicted in Fig. 1, where each circle in the figure represents a processor and each four-tuple of processors represents a system configuration. The first configuration in the execution is the top leftmost four-tuple (marked c_1) and the last is the bottom leftmost four-tuple (marked c'_1).

The value of the clock of the processor P_i is written inside the circle that represents P_i. Each processor P_i has one outgoing arrow which indicates the next processor that P_i is about to read from. If P_i is marked with a box (inside the circle) then P_i is about to execute a step during the next pulse (otherwise, there is no change in the state of P_i). The execution includes six pulses, during which P_1 executes six steps and every other processor executes three steps.

Two configurations c and c' are *equivalent* if there is some constant l such that c' is derived from c by adding l to every *clock* in c. (Everything else is the same about the states, including each processor's choice of which processor to read from next.) Note that c_1 and c'_1 in Fig. 1 are equivalent with $l = 6$. To construct E we have to show that there is a configuration c' equivalent to c_1, that is reachable from the initial configuration c_0 of the system.

Starting from c_0, we can reach a configuration c in which $P_4.clock < P_3.clock < P_1.clock < P_2.clock$ and the arrows are in the same direction as in c_1 by first activating only P_4 till it is about to read from P_2, then activating only P_3 till $P_3.clock > P_4.clock$ and P_3 is about to read

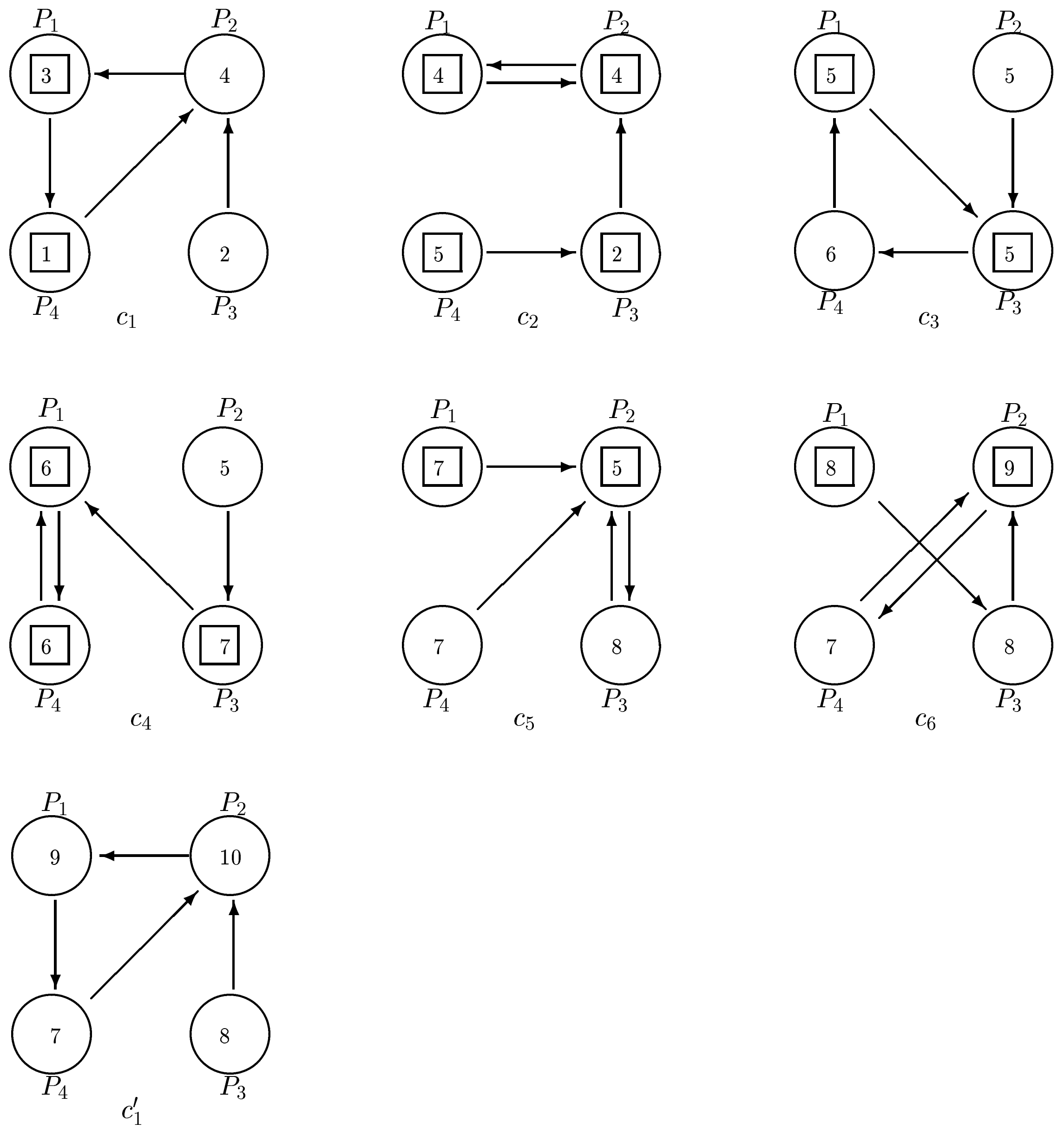

Figure 1: Execution of First Shot Algorithm

from P_2, then activating only P_1 till $P_1.clock > P_3.clock$ and P_1 is about to read from P_4, and finally activating only P_2 till $P_2.clock > P_1.clock$ and P_2 is about to read from P_1.

Although the relation $P_4.clock < P_3.clock < P_1.clock < P_2.clock$ holds in c, c may not be equivalent to c_1: for instance, in c it might be true that $P_4.clock - P_3.clock > 1$. However, it is easy to check that when the system is started with c and the processors are activated in the same order as in Fig. 1, then after six pulses a configuration that is equivalent to c_1 is reached.

In the execution that appears in Fig. 1, P_1 does not find the maximal clock value (i.e., the value 10 in c'_1). Moreover, since c_1 and c'_1 are equivalent, there is an execution that starts with c'_1 and ends with equivalent configuration c''_1 in which P_1 does not hold the maximal value either, and so on and so forth till the execution includes more than k pulses. After the execution includes more than k pulses we activate all the processors in each pulse and then P_1 adjusts its clock to the maximal value. Note that this last operation violates the adjustment requirement.

3.2 Blind Writes

This section shows that no self-stabilizing wait-free clock synchronization algorithm is possible if processors can start with incorrect information about the values of their own shared variables. This result motivates our assumption that the state of the processor includes the value in its shared register, for any self-stabilizing wait-free clock synchronization algorithm.

A self-stabilizing algorithm is *blind-write* if the value of its shared variable is not part of its state. Note that in a non-self-stabilizing algorithm, a processor can accurately keep track of the last value written to its own variable, and thus it will not be blind-write.

Theorem 3.1 *There is no blind-write self-stabilizing wait-free clock synchronization algorithm.*

Proof: Assume towards a contradiction that there is such an algorithm $\mathcal{A}$ with convergence time k. Suppose the system has three processors, P, Q, and R, with shared variables $P.v$, $Q.v$, and $R.v$ respectively.

Lemma 3.2 *In any infinite execution of $\mathcal{A}$ in which only P works, P must read $R.v$ infinitely often.*

Proof: Suppose not. Consider execution E_P in which Q and R nap, P works and P reads $R.v$ only finitely often. Let c be the k-th configuration after P's last read of $R.v$. Let s be P's state in c, x be $P.v$'s value in c, and y be $Q.v$'s value in c. Let m be the value of P's clock in x. By the adjustment condition, P must increment its clock by 1 at every pulse after c.

Let E_R be an infinite execution that begins with P in state s, $P.v$ holding x, and $Q.v$ holding y. Furthermore, suppose that P and Q nap and R works. By the adjustment condition, eventually R's clock reaches some value $m' > m$. Truncate E_R k pulses after this point. Call the result E'_R.

Extend E'_R to infinite execution $E'_R E_{PR}$ by having Q continue to nap and P and R work. Note that in E_{PR}, R must increment its clock by 1 at every pulse, by the adjustment condition. P goes through the same sequence of state transitions in E_{PR} as P does in the suffix of E_P after c. Thus P increments its clock by 1 at every pulse. But since m and m' are not equal, P's and R's clocks are never equal in E_{PR}, violating the agreement condition. ■

Let E_P be an infinite execution in which P works and Q and R nap. Let c be a configuration that follows the first k pulses of E_P in which P is about to read $R.v$. Such a configuration exists by the lemma. Let s be P's state in c and v be the value of $P.v$ in c. Let m be the clock value that P writes in the next step. (By the adjustment condition, P has no choice concerning m.) Let c' be next configuration after c in which P is about to read $R.v$. Again, such a configuration exists by the lemma. Let s' be P's state in c' and v' be the value of $P.v$ in c'. Let m' be the clock value that P writes in the next step. Note that $R.v$ has a fixed value, say z, throughout E_P since R naps.

Let E_Q be the same as E_P except that P and Q reverse roles, i.e., the initial states of P and Q are switched, the initial values of $P.v$ and $Q.v$ are switched, P naps and Q works. Let d' be the configuration in E_Q that is the analog of configuration c' in E_P.

Let E be an execution that begins with P in state s, Q in state s', $R.v$ equal to z, in which P and Q work and R naps. In the first pulse, P does the same thing that it does after c in E_P and Q does the same thing that it does after d' in E_Q. Thus the clocks of P and Q are not equal. By the agreement condition, there is some point in E at which either P or Q adjusts its clock (i.e., changes it in a way other than incrementing by 1).

Case 1: P adjusts its clock in E. Then we can construct an execution by truncating E_P at c, changing Q's state in the initial configuration to be s', and then extending the execution to mimic E. In the extension, P will mimic its behavior in E; in particular it will adjust its clock. But P has already been working for more than k pulses, violating the adjustment condition.

Case 2: Q adjusts its clock in E. Then we can construct an execution by truncating E_Q at d', changing P's state in the initial configuration to be s, and then extending the execution to mimic E. In the extension, Q will mimic its behavior in E; in particular it will adjust its clock. But Q has already been working for more than k pulses, violating the adjustment condition. ■

4 In-Phase Algorithms

In this section we present two algorithms.

4.1 $O(n^3)$ Algorithm

In this section we describe a self-stabilizing wait-free clock synchronization algorithm with $k = O(n^3)$. The main observation leading to our solution is the following: In the first shot algorithm a processor that misbehaves needs to frequently stop and resume operation in order to mislead a working processor. Thus we choose to "punish" a processor that naps by excluding it from the "game" for a long enough time. We design a mechanism that provides a napping processor an indication on the fact it has been napping. When a processor receives such an indication it stops participating for a long enough period.

A processor P *stops increasing* its clock if P increases its clock in one pulse and does not increase its clock in the next pulse. If during the execution of the first shot algorithm there is an interval of $2(n-1)$ pulses during which no processor stops increasing its clock, then every working processor finds the maximal clock value in the system. The reason is that the maximal clock value is held by at least one increasing processor after the first $n-1$ pulses and is held by every increasing processor after the second $n-1$ pulses. To use the above observation we design a mechanism by which a processor can tell whether it stopped increasing or not. Whenever a processor P gets an indication that it stopped increasing, it continues not to increase for a certain number of steps. The number of steps is chosen to be long enough so that P does not mislead increasing processors.

The algorithm for processor P_j is shown in Fig. 2. Informally, the algorithm works as follows. P_j reads the *clock* and *count* variables of its neighbors one after the other in cyclic order. Each step is started with such a read operation (line 4), say from P_i. Then P_j increments its own *count* variable by 1. Next P_j computes the number of steps that P_i executed since the last read of P_j from P_i using the variable *previous* (lines 6,7). If P_j realizes that P_i executed more than $n-1$ steps between those reads, then P_j concludes that P_j omitted at least one step and assigns $wait := 4n^3$ (line 9). (Note that by the nature of the faults to be tolerated by a self-stabilizing algorithm, that conclusion could be wrong the first time, since a processor can be initiated with arbitrary values of *previous*.) P_j decrements *wait* by 1 in every step until $wait = 0$ (line 8). As long as $wait > 0$, P_j does not change the value of *clock* (line 10). When $wait = 0$, P_j chooses the greater of its own clock and P_j's clock and increments it by 1 to obtain its new clock value. At the end of each step, P_j writes the new values of *clock* and *count*.

We now prove the algorithm is correct. Let E be an arbitrary execution.

Lemma 4.1 *For all times T and all processors P, if $work(P,E,T) \geq 4n^3 + n - 1$, then P increases its clock at step T.*

Proof: Let $l = work(P,E,T)$. During the first $n-1$ (out of l) successive steps of P, P reads all the *count* variables in the system and any further computation of *delta* results in a value that is less than or equal to $n-1$. Thus following those first $n-1$ steps, P decrements *wait* until it reaches the value 0. The lemma follows. ■

```
do forever
  for $i := 1$ to $n$ (not including $j$)
    do
      read($P_i.clock, P_i.count$)
      $count := count + 1$
      $delta := P_i.count - previous[i]$
      $previous[i] := P_i.count$
      if $wait \geq 1$ then $wait := wait - 1$
      if $delta > n - 1$ then $wait := 4n^3$
      if $wait = 0$ then $clock := max(P_i.clock, clock) + 1$
      write($clock, count$)
    od
od
```

Figure 2: Program of P_j for $O(n^3)$ In-Phase Algorithm

Definition 4.1 *$stop\_increasing(P, E)$ is defined to be the number of pulses in E such that (1) P's clock does not change during the pulse, and (2) P's clock does change during the preceding pulse.*

It is sufficient for our correctness proof to consider an arbitrary sub-execution E_q of E during which some processor Q executes all the pulses, and the number of pulses is greater than $k = 8n^3 + n - 1$. We will show that in E_q the adjustment condition holds for Q and the agreement condition holds for Q and any other processor.

Let E'_q be the sub-execution of E_q that begins after the first $4n^3 + n - 1$ pulses of E_q and lasts for $4n^3$ pulses. Note that by Lemma 4.1, Q increases its clock at every step of E'_q.

Lemma 4.2 *For any processor P, $stop\_increasing(P, E'_q) \leq 2(n-1)$.*

Proof: A processor P may stop increasing only if P assigns $wait := 4n^3$ or P fails, i.e., does not execute a step. Since E'_q includes only $4n^3$ pulses, P may stop increasing at most once due to the assignment of $wait := 4n^3$. Thus, we have to count the maximal number of times that a processor P may stop increasing due to failures. If P takes fewer than $2(n-1)$ steps in E_q, then we are done.

Suppose P takes at least $2(n-1)$ steps. During one of the first $n-1$ of those steps, P reads $Q.count$. If P fails once more following this read, then the next time P reads $Q.count$, P finds out that $delta > n - 1$ and assigns $wait := 4n^3$. Hence, we have to count the maximal number of times P may stop increasing before that read operation and assignment occurs. P

may stop increasing after every single step that it takes, if for instance it skips every other pulse. Thus, P may additionally stop increasing at most $n-2$ times.

To summarize, in the worst case P stops increasing due to napping after each of the first $n-1$ read operations (in the last of those read operations P reads $Q.count$). Then P stops increasing due to napping after each of the next $n-2$ read operations (before P gets to read $Q.clock$ again). At last P stops increasing once more due to the assignment of $wait := 4n^3$ (right after reading $Q.clock$). ∎

Lemma 4.3 *There is a sub-execution $\hat{E}_q$ of E'_q with at least $2(n-1)$ pulses such that for every processor P, $stop\_increasing(P, \hat{E}_q) = 0$.*

Proof: The lemma is proved by the pigeon-hole principle. By Lemma 4.2, each processor $P \neq Q$ may stop increasing at most $2(n-1)$ times. Thus there are at most $2(n-1)^2$ stop-increasing events during E'_q. The number of intervals of E'_q delimited by either the begining of E'_q, or a stop-increasing event, or the end of E'_q (not containing a stop-increasing event) is $2(n-1)^2+1$. Since E'_q contains $4n^3$ pulses, there is a sub-execution of E'_q containing at least $4n^3/(2(n-1)^2+1) > 2(n-1)$ pulses in which no processor stops increasing. ∎

A processor R holds the *maximal clock value* in a certain configuration c if $R.clock(c) \geq P.clock(c)$ for all processors P.

Lemma 4.4 *In the last configuration of $\hat{E}_q$, $Q.clock$ holds the maximal clock value.*

Proof: By Lemma 4.3, $stop\_increasing(P, \hat{E}_q) = 0$ for every P. Either P does not increase its clock during $\hat{E}_q$ at all or after the first time P increases its clock P continues to increase its clock in later pulses. We say that after the first increase of P, P is *increasing.* During each clock pulse every processor may either (a) not increase its clock or (b) increment its clock by 1 or (c) assign its clock to be the value of the clock of another processor + 1. Therefore, once the clock of an increasing processor has the maximal clock value in the system, this clock is incremented by 1 in every clock pulse and thus it stays the maximal clock value in the system.

Let $\mathcal{P}(c)$ be the set of processors with the maximal clock value in a system configuration c. Let c be the the first configuration of $\hat{E}_q$. If there is no increasing processor in $\mathcal{P}(c)$ then within the first $n-1$ pulses at least one increasing processor reads a value from a processor in $\mathcal{P}$ and assigns its clock to be a greater value. (Notice that during $\hat{E}_q$, Q is an increasing processor.) Just after that assignment the set $\mathcal{P}$ includes an increasing processor. Once there is an increasing processor P in $\mathcal{P}$, then within the next $n-1$ pulses, Q reads P's clock value and assigns its clock to be the maximal value too. Thus, after the first $2(n-1)$ pulses of $\hat{E}_q$, Q has the maximal clock value. ∎

Let c_q be the first configuration in E_q for which Q holds the maximal clock value. Note that the existence of such a configuration is proven in Lemma 4.4.

Lemma 4.5 *For every configuration in E'_q that follows c_q, Q holds the maximal clock value.*

Proof: Throughout E'_q, $Q.wait = 0$. Thus in any step during E'_q, Q may (a) increment its clock by 1 or, in case Q does not hold the maximal clock value, Q may (b) set its clock to some value that Q reads from the clock of another processor, and increment this value by 1. Hence, once Q holds the maximal clock value, Q does not execute (b) above for the rest of E'_q. ■

Theorem 4.6 *The above algorithm is a wait-free clock synchronization algorithm with convergence time $k = 8n^3 + n - 1$.*

Proof: By Lemmas 4.4 and 4.5, Q has the maximal clock value after $\hat{E}_q$. Thus the only change ever made to $Q.clock$ is to increment it by 1, as long as Q continues to work. Hence, the adjustment requirement holds. If there is any other processor P that has been working long enough, then it too has the maximal clock value, which is equal to Q's clock value. Therefore the agreement requirement holds too. ■

During the proof we did not restrict the values of the variables (*clock*, *count*, *previous*, *wait*) in the initial configuration. Thus the system could be initiated with any possible configuration and still fulfill the adjustment and agreement requirements. In other words the above algorithm is also self-stabilizing. Note that the algorithm presented in this section as well as the algorithms presented in the sequel assume that clock values are unbounded.

4.2 $O(n^2)$ Algorithm

In order to reduce k we would like to limit the number of times a processor may "disturb" the system before it finds out that it was napping. To do so we let each processor adjust its clock only once every $n-1$ pulses. Roughly speaking, a processor "remembers" in a local variable the maximal clock value it read and uses it to adjust its clock once every $n-1$ pulses. The drawback of this approach is that it violates the self-stabilization property: A napping processor might be initiated with a very large value for the local variable that stores the maximal clock value observed and then this very large value could be used at a most unfortunate time, causing working processors to adjust clocks. Thus, our next algorithm achieves better performance at the cost of the self-stabilization property.

Fig. 3 describes the wait-free clock synchronization algorithm for $k = O(n^2)$. Since the algorithm is not self-stabilizing, we must define the initial values of the variables. For all i and j, we assume that the initial values of $P_j.count$, $P_j.previous[i]$, $P_j.clock$, $P_j.max\_clock$ and $P_j.wait$ are all 0.

Informally, the nature of the variables and operations is the same as in the previous algorithm. The main difference from the previous algorithm is that P_j does not adjust (increase by more than 1) its clock immediately after reading a greater clock value. P_j may adjust its clock only in one step out of every $n-1$ successive steps (line 12). In the rest of the steps, P_j just increments its clock by 1 (line 5). To keep track of the maximal clock value, P_j updates its internal variable *max_clock* (line 11).

Roughly speaking, the improvement upon the previous algorithm is gained by the fact that a processor P that does not work correctly may "disturb" the system (i.e., increase its clock by more than one) only once every $n-1$ steps. This helps to reduce the number of times P can "disturb" the system before finding out that it has to wait. A processor P uses the local variable *max_clock* to keep track of the maximal clock value it observed. Only after P reads from every other processor does P set its clock to the value of *max_clock*. Note that this algorithm is not self-stabilizing, since in a self-stabilizing algorithm *max_clock* could be arbitrarily initialized.

```
01 do forever
02   for $i := 1$ to $n$ (not including $j$)
03     do
04       read($P_i.clock, P_i.count$)
05       $clock := clock + 1$
06       $count := count + 1$
07       $delta := P_i.count - previous[i]$
08       $previous[i] := P_i.count$
09       if $wait \geq 1$ then $wait := wait - 1$
10       if $delta > n - 1$ then $wait := 8n^2$
11       if $wait = 0$ then $max\_clock := max(max\_clock, P_i.clock) + 1$
12       if $wait = 0$ and ($i = n$ or ($j = n$ and $i = n - 1$)) then $clock := max\_clock$
13       if $wait \neq 0$ then $max\_clock := clock$
14       write($clock, count$)
15     od
16 od
```

Figure 3: Program of P_j for $O(n^2)$ In-Phase Algorithm

The outline of the proof is the same as that for the previous algorithm. We consider any sub-execution that contains at least $k = 16n^2 + n - 1$ pulses and during which there is at least one processor, say Q, that executes every clock pulse. We use the pigeon-hole principle to deduce that there is a sub-execution of $4(n-1)$ pulses satisfying certain requirements. Thanks to these requirements, we can show that Q finds the maximal clock value at the end of this sub-execution. Once this happens, Q satisfies the adjustment and agreement conditions.

Definition 4.2 *$g\_clock(c)$ is the maximal value of P.clock, over all P, in configuration c.*

Definition 4.3 *$g\_max\_clock(c)$ is the maximal value of P.max_clock, over all P, in configuration c.*

Lemma 4.7 *For any processor P and for every configuration c in an initialized execution, $P.max\_clock(c) \geq P.clock(c)$ and hence $g\_max\_clock(c) \geq g\_clock(c)$.*

Proof: The lemma is proved by induction on the steps of P. The base case is by the fact that in c_0 (the initial configuration), $P.max\_clock(c_0) = P.clock(c_0) = 0$. For the inductive case: During any step of P, P increments *clock* by 1 and then either increases *max_clock* by at least 1 or makes the values of *clock* and *max_clock* equal. ■

Lemma 4.8 *Consider any initialized execution. For every configuration c that immediately follows the execution of $l > 8n^2 + n - 1$ successive steps by P, $P.wait(c) = 0$.*

Proof: During the first $n - 1$ (out of l) successive steps of P, P reads all the *count*s in the system and any subsequent computation of *delta* results in a value that is less than or equal to $n - 1$. Thus following those first $n - 1$ steps, P decrements *wait* till it reaches the value 0. The lemma follows. ■

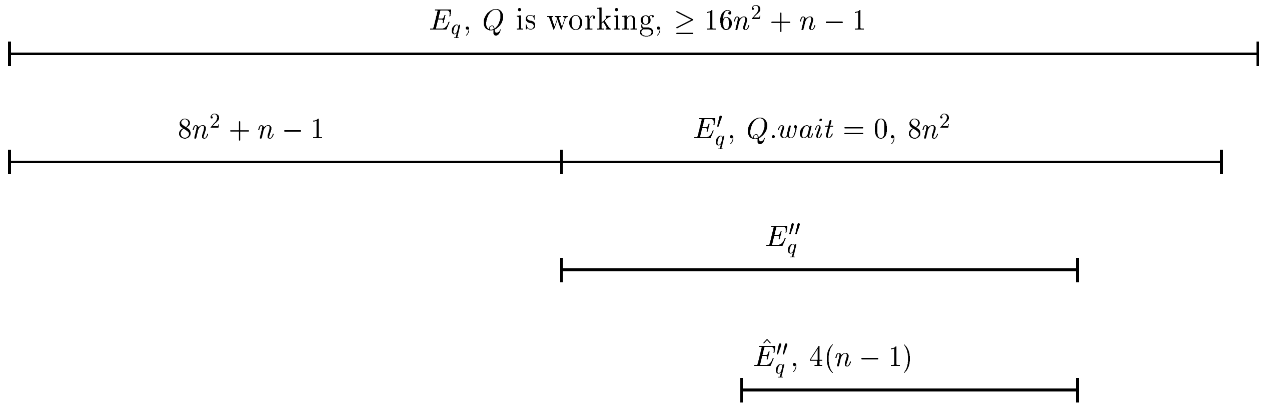
E_q, Q is working, $\geq 16n^2 + n - 1$
$8n^2 + n - 1$
E'_q, $Q.wait = 0$, $8n^2$
E''_q
$\hat{E}''_q$, $4(n-1)$

Figure 4: Definitions of sub-executions within E_q

Definition 4.4 *For execution E, define $working(P, E)$ to be true if P executes all the steps during E.*

Definition 4.5 *Let E' be a suffix of execution E. Define $first\_stop(P, E, E')$ to be true if the first time that P does not execute a step in E is during E'; otherwise $first\_stop(P, E, E')$ is false.*

Definition 4.6 *Let E' be a suffix of execution E. Define $disturbing(P, E, E')$ to be 0 if $working(P, E)$ is true. Otherwise if $working(P, E)$ is false, i.e., P missed at least one step during E, let a be the first step P missed during E, and define $disturbing(P, E, E')$ to be the number of pulses in E' and after a, at which P increases $P.clock$ by more than 1.*

It is sufficient for our proof to consider an arbitrary sub-execution E_q of an arbitrary initialized execution E during which some processor Q executes all the steps, and the number of steps is greater than or equal to $k = 16n^2 + n - 1$. We show, in this case, that Q will satisfy the adjustment and agreement conditions.

Let E'_q be the sub-execution of E_q that begins after the first $8n^2 + n - 1$ pulses of E_q and lasts for $8n^2$ pulses. Note that, by Lemma 4.8, $Q.wait = 0$ throughout E'_q. Refer to Fig. 4 throughout the remainder of the correctness proof.

Lemma 4.9 *For all processors P, $disturbing(P, E'_q, E'_q) \leq 1$.*

Proof: A processor may increase its clock by more than 1 at most once every $n - 1$ steps.

The proof considers two cases. First we consider any processor P that does not read $Q.count$ during the $n - 1$ pulses of E_q that precede E'_q (note that E'_q is a sub-execution of E_q that begins after the first $8n^2 + n - 1 > n - 1$ pulses of E_q). During those $n - 1$ pulses of E_q, $Q.count$ is increased by $n - 1$, thus if P reads $Q.count$ after those $n - 1$ pulses (i.e., during E'_q), P finds out that $delta > n - 1$ and assigns $wait := 8n^2$. Since P executes line (12) once between each pair of reads of a particular other processor's variables, it holds that before that first read (and assignment) P may increase its clock by more than 1 at most once. Moreover, following that first read (and assignment) P does not increase $P.clock$ by more than 1 for the rest of E'_q.

We have to consider a processor P that reads $Q.count$ during the $n - 1$ pulses of E_q that precede E'_q. Following that read operation if P misses even a single execution of a step P has to discover that $delta > n - 1$ in the successive read operation from $Q.count$. Again since P executes line (12) once between each pair of reads of $Q.count$, it must hold that before P discovers that $delta > n - 1$ and assigns $wait := 8n^2$, P may increase its clock by more than 1 only once. Note that for a processor P that does not miss any step during E'_q it holds that $disturbing(P, E'_q, E'_q) = 0$. ■

Lemma 4.10 *There exists a prefix E''_q of E'_q that has suffix $\hat{E}''_q$ consisting of $4(n - 1)$ steps such that for every processor P, $first\_stop(P, E''_q, \hat{E}''_q) = false$ and $disturbing(P, E''_q, \hat{E}''_q) = 0$.*

Proof: The lemma is proved by the pigeon-hole principle. During the $8n^2$ steps of E'_q each processor $P \neq Q$ may stop working for the first time at most once and by Lemma 4.9, P may disturb the system thereafter at most once. Thus the sum of the first-stop and disturbing events during the first $8n^2$ pulses of E'_q is at most $2(n - 1)$. Hence, there is a sub-execution

$\hat{E}''_q$ of E'_q consisting of at least $8n^2/(2(n-1)+1) > 4(n-1)$ pulses in which no processor first-stops or disturbs. Let E''_q be the prefix of E'_q through the end of $\hat{E}''_q$. ■

Lemma 4.11 *Let $\hat{c}$ be the last configuration in $\hat{E}''_q$. Then $Q.max\_clock(\hat{c}) = Q.clock(\hat{c}) = g\_max\_clock(\hat{c})$.*

Proof: Let $\mathcal{P}$ be the set of processors P such that $working(P, E''_q) = true$. First we show that after the first $n-1$ pulses of $\hat{E}''_q$, every processor P in $\mathcal{P}$ increases $P.max\_clock$ at every step it takes through the end of $\hat{E}''_q$. Suppose P has $P.wait > 0$ at some point after the first $n-1$ pulses of $\hat{E}''_q$. This can only be because P decides it has to wait (based on computing $delta > n-1$) during those first $n-1$ pulses or earlier. By the code, once P sets $P.wait$ equal to $8n^2$, as long as $P.wait$ is bigger than 0, $P.max\_clock$ and $P.clock$ are equal to each other and are incremented by 1 in every step. Once P stops waiting (if it ever does during $\hat{E}''_q$), it continues increasing $P.max\_clock$. If P never has $P.wait > 0$ after the first $n-1$ pulses of $\hat{E}''_q$, then by the code it always increases $P.max\_clock$.

During the first $n-1$ steps of Q in $\hat{E}''_q$, Q reads the clock of every processor R such that $R \notin \mathcal{P}$. By the definition of $\hat{E}''_q$, it holds that $first\_stop(R, E''_q, \hat{E}''_q) = false$. Since $R \notin \mathcal{P}$ it must hold that R does not execute at least one step during E''_q and before $\hat{E}''_q$. Moreover, by the definition of $\hat{E}''_q$ it holds that $disturbing(R, E''_q, \hat{E}''_q) = 0$. Hence, during $\hat{E}''_q$, R increases $R.clock$ by at most 1 in every step, while Q increases $Q.max\_clock$ by at least 1 in every step. Thus, any successive read of Q from $R.clock$ does not change the value of $Q.max\_clock$. In other words, in every configuration of $\hat{E}''_q$ that follows the first $n-1$ steps of $\hat{E}''_q$, it holds that $Q.max\_clock \geq R.clock$. Let c be the configuration that immediately follows the first $n-1$ pulses in $\hat{E}''_q$. Let $W \in \mathcal{P}$ be a processor such that for every other processor $P \in \mathcal{P}$, $W.max\_clock(c) \geq P.max\_clock(c)$. Note that since $W \in \mathcal{P}$, then between c and $\hat{c}$, W increments $W.max\_clock$ by 1 in every clock pulse and $W.max\_clock \geq P.max\_clock$.

During the second $n-1$ pulses, W assigns to its clock the value of $W.max\_clock$. Thus in the configuration c' that follows $2(n-1)$ pulses of E''_q, $W.max\_clock(c') = W.clock(c') = W.max\_clock(c) + n - 1$. Moreover, following that assignment and throughout $\hat{E}''_q$, $W.clock = g\_clock$. Before the third set of $n-1$ pulses elapses, Q reads the value of $W.clock$. Thus in the configuration c'' that follows the first $3(n-1)$ pulses of $\hat{E}''_q$, it holds that $Q.max\_clock(c'') = W.max\_clock(c'') = W.clock(c'') = W.max\_clock(c) + 2(n-1)$ and for every processor $P \in \mathcal{P}$, $Q.max\_clock(c'') \geq P.max\_clock(c'')$ and in $\hat{c}$ $Q.clock(\hat{c}) = Q.max\_clock(\hat{c})$.

Thus, in order to show that $Q.max\_clock(\hat{c}) = g\_max\_clock(\hat{c})$, we only have to show that for every processor $R \notin \mathcal{P}$, $Q.max\_clock(\hat{c}) \geq R.max\_clock(\hat{c})$. Assume toward contradiction that $R.max\_clock(\hat{c}) > Q.max\_clock(\hat{c})$. Since $Q.max\_clock(\hat{c}) = W.clock(\hat{c}) = g\_clock(\hat{c})$, then $R.clock(\hat{c}) < R.max\_clock(\hat{c})$. $R.max\_clock(\hat{c})$ may be greater than $R.clock(\hat{c})$ only due to a value R reads from a processor clock, say $P.clock$, during a step r that is one of the last $n-2$ steps of R before $\hat{c}$. (Recall that during at most $n-1$ successive steps of R there is a step in which $R.clock = R.max\_clock$.)

Consider two cases: (a) r occurs after the first $2(n-1)$ steps of $\hat{E}''_q$, or (b) r occurs during or before the first $2(n-1)$ steps of $\hat{E}''_q$. In case (a), R assigns to *R.max_clock* a value that is less than or equal to *W.clock* by definition of W.

In case (b), R does not execute at least $2(n-1)-(n-2)$ steps (otherwise, R assigns *R.clock* := *R.max_clock* after r which contradicts the definition of r). Let c_r be the configuration that immediately follows r. Then $R.max\_clock(c_r) \leq g\_clock(c_r)+1$. As mentioned above, R may increase the value R read by at most $n-2$ before R writes (by choice of r). Thus, $R.max\_clock(\hat{c}) \leq R.max\_clock(c_r)+n-2$. Moreover, since during the last $2(n-1)$ steps W increases its clock (and hence increases *g_clock*) by at least $2(n-1)$, then $g\_clock(c_r) \leq g\_clock(\hat{c})-2(n-1)$. Hence, $R.max\_clock(\hat{c}) \leq R.max\_clock(c_r)+n-2 \leq g\_clock(c_r)+1+n-2 \leq g\_clock(\hat{c})-2(n-1)+1+n-2 < g\_clock(\hat{c})$. ■

Lemma 4.12 *In every configuration c' in E_q after $\hat{c}$, $Q.max\_clock(c') = Q.clock(c') = g\_max\_clock(c')$.*

Proof: We prove the lemma by induction on the number of pulses in E_q following $\hat{c}$. Let p be a pulse that immediately follows a configuration c_i in which $Q.max\_clock(c_i) = Q.clock(c_i) = g\_max\_clock(c_i)$. During p, Q increments *Q.max_clock* and *Q.clock* by 1, and every other processor P may increase *P.max_clock* and *P.clock* to be at most the same value as *Q.max_clock*. Thus, in the configuration c_{i+1} that follows p it holds that $Q.max\_clock(c_{i+1}) = Q.clock(c_{i+1}) = g\_max\_clock(c_{i+1})$. ■

Theorem 4.13 *The above algorithm is a wait-free clock synchronization algorithm with convergence time $k = 16n^2+n-1$.*

Proof: By Lemmas 4.12 and 4.7, Q has the maximal clock value after $\hat{E}''_q$. Thus the only change ever made to *Q.clock* is to increment it by 1, as long as Q continues to work. This implies the adjustment requirement. If there is any other processor P that has been working long enough, then it too has the maximal clock value, which is equal to Q's clock value. Thus the adjustment requirement holds too. ■

5 Out-of-Phase Algorithm

In this section we assume that there is no common clock pulse. Each processor owns an independent pulse generator. Each processor P may execute a step at any time, subject to the restriction that the time elapsed between two successive steps of P is greater than or equal to a single time unit. Since read and write operations may require some duration in time, it is not natural to assume that in a single step a processor P atomically reads a shared variable, modifies its own state according to the value read and writes a value in its shared variable. Instead, we assume that a clock pulse triggers P to execute simultaneously both write operation to its own variable and read of a shared variable. Thus, the value written during a pulse is the function of the processor state before the pulse (and not of the value the processor reads during the pulse).

A processor works *as soon as possible* (abridged asap) if the time elapsed between two successive steps of the processor is exactly a single time unit.

The code for processor P_j in our out-of-phase algorithm appears in Fig. 5. We assume for any two processors P_j and P_i that the initial values of $P_j.previous[i]$, $P_j.clock$, $P_j.max\_clock$ and $P_j.wait$ are all 0. Informally, the nature of the variables and operations is the same as in the previous algorithms. The main difference from the previous algorithm is that whenever P_j reads a clock value that is greater than max_clock, P_j assigns max_clock to be the value P_j read without incrementing it by 1 (line 13).

```
01 do forever
02   for $i := 1$ to $n$ (not including $j$)
03     do
04       write($clock + 1, count + 1$)
05       read($P_i.clock, P_i.count$)
06       $max\_clock := max\_clock + 1$
07       $clock := clock + 1$
08       $count := count + 1$
09       $delta := P_i.count - previous[i]$
10       $previous[i] := P_i.count$
11       if $wait \geq 1$ then $wait := wait - 1$
12       if $delta > n - 1$ then $wait := 12n^2$
13       if $wait = 0$ then $max\_clock := max(max\_clock, P_i.clock)$
14       if $wait = 0$ and $(i = n$ or $(j = n$ and $i = n - 1))$ then $clock := max\_clock$
15       if $wait \neq 0$ then $max\_clock := clock$
16     od
17 od
```

Figure 5: Program of P_j for $O(n^2)$ Out-of-Phase Algorithm

The correctness proof is similar to the correctness proof of the previous algorithm. Instead of arguing about the number of steps that Q executes successively (as in the previous proof), we argue about a sub-execution during which Q executes all its steps asap. Therefore between any two successive read operations by Q of another processor's clock, P's count is increased by at most $n - 1$. Thus, following the first sequence of read operations by Q, whenever Q computes $delta$, its value is less than or equal to $n - 1$. We consider a time period in which $Q.wait = 0$, and concentrate on a period of $12n^2$ pulses during which every processor that assigns $wait := 12n^2$ does not have enough time to decrement it to 0. We show that during the

$12n^2$ successive pulses, each processor may "disturb" the execution at most three times. Thus there is a sub-execution of length $4n$ pulses with no "disturbances", ensuring that Q finds the maximal clock value by the end.

A processor P is working *asap relative to* Q if between any two successive steps of Q there is a step of P.

Definition 5.1 *For execution E, define $working(P, E, Q)$ to be true if, during E, P executes all its steps asap relative to Q.*

Definition 5.2 *Let E' be a suffix of execution E that starts immediately following a step of Q. Define $first\_stop(P, E, E', Q)$ to be true if the first time that P does not execute a step asap relative to Q is during E'; otherwise $first\_stop(P, E, E', Q)$ is false.*

Definition 5.3 *Let E' be a suffix of execution E that starts immediately following a step of Q. Define $first\_wait(P, E, E', Q)$ to be true if the first time that P assigns $wait := 12n^2$ is during E'; otherwise $first\_wait(P, E, E', Q)$ is false.*

Definition 5.4 *Let E' be a suffix of execution E that starts immediately following a step of Q. Define $disturbing(P, E, E', Q)$ to be 0 if $working(P, E, Q)$ is true. Otherwise if $working(P, E, Q)$ is false, i.e., P missed at least one step during E relative to Q, let t be the first time P missed such a step during E and define $disturbing(P, E, E', Q)$ to be the number of pulses in E' and after t, at which P increases $P.clock$ by more than 1.*

It is sufficient for our proof to consider an arbitrary sub-execution E_q of an arbitrary initialized execution E during which some processor Q executes all the steps asap and the number of steps is more than $k = 24n^2 + n$. We show in this case that the adjustment condition holds for Q and that the agreement condition holds for Q and any other relevant processor.

Let E'_q be the sub-execution of E_q that begins after the first $12n^2 + n - 1$ asap steps of Q and lasts for $12n^2$ asap steps of Q. Note that $Q.wait = 0$ throughout E'_q.

Lemma 5.1 *For every processor P, $disturbing(P, E'_q, E'_q, Q) \leq 1$.*

Proof: A processor may increase its clock by more than 1 at most once every $n - 1$ steps.

The proof considers two cases. First we consider any processor P that does not read $Q.count$ during the $n - 1$ pulses of E_q that precede E'_q (note that E'_q is a sub-execution of E_q that does not include the first at least $12n^2 + n - 1 > n - 1$ steps of Q). During those $n - 1$ pulses of E_q, $Q.count$ is increased by $n - 1$; thus if P reads $Q.count$ after those $n - 1$ pulses (i.e., during E'_q) P finds out that $delta > n - 1$ and assigns $wait := 12n^2$. Since P executes line 14 once between each pair of reads of a particular other processor's variables, before that first read (and assignment) P may increase its clock by more than 1 at most once. Moreover, following that first read (and assignment) P does not increase *P.clock* by more than 1 for the rest of E'_q, since it has $P.wait > 0$ for the rest of E'_q.

We have to consider a processor P that does read *Q.count* during the $n-1$ pulses of E_q that precede E'_q. Following that read operation if P does not work asap relative to Q at least once then P has to discover that $delta > n-1$ in the successive read operation from *Q.count*. Again since P executes line 14 once between each pair of reads of a particular other processor's variable, before P discovers that $delta > n-1$ and assigns $wait := 12n^2$, P may increase its clock by more than 1 only once. ■

Lemma 5.2 *There exists a prefix E''_q of E'_q that has a suffix $\hat{E}''_q$ consisting of $4n$ steps of Q such that for every processor P, $first\_stop(P, E''_q, \hat{E}''_q, Q) = false$, $first\_wait(P, E''_q, \hat{E}''_q, Q) = false$ and $disturbing(P, E''_q, \hat{E}''_q, Q) = 0$.*

Proof: The lemma is proved by the pigeon-hole principle. During the $12n^2$ steps of E'_q each processor $P \neq Q$ may stop working for the first time at most once, may start waiting for the first time at most once and by Lemma 5.1, P may disturb the system at most once. Thus, there is a sub-execution $\hat{E}''_q$ of E'_q consisting of at least $12n^2/(3(n-1)+1) > 4n$ steps of Q in which none of the above occurs. Let E''_q be the prefix of E'_q through the end of $\hat{E}''_q$. ■

Lemma 5.3 *Let $\hat{c}$ be the last configuration in $\hat{E}''_q$. Then $Q.max\_clock(\hat{c}) = Q.clock(\hat{c}) = g\_max\_clock(\hat{c})$. ($g\_max\_clock$ is defined the same as for the in-phase system.)*

Proof: Let $\mathcal{P}$ be the set of processors P such that $working(P, E''_q, Q) = true$. Note that because $first\_stop(R, E''_q, \hat{E}''_q, Q) = false$, every processor $R \notin \mathcal{P}$ must have not executed a step asap relative to Q during E''_q and before $\hat{E}''_q$.

During the first $n-1$ steps by Q in $\hat{E}''_q$, Q reads the clocks of all processors R such that $R \notin \mathcal{P}$. By Lemma 5.2, $disturbing(R, E''_q, \hat{E}''_q) = 0$. Thus during $\hat{E}''_q$, R increases *R.clock* by at most 1 in every step while Q increases *Q.max_clock* by at least 1 in every step. Thus, any successive read of Q from *R.clock* does not influence the value of *Q.max_clock*.

Denote the configuration that immediately precedes the i'th step of Q during $\hat{E}''_q$ by b_i and denote the configuration that immediately follows the i'th step of Q by c_i.

Let $W \in \mathcal{P}$ be a processor such that for every other processor $P \in \mathcal{P}$, $W.max\_clock(b_{n-1}) \geq P.max\_clock(b_{n-1})$ (note that W may be Q).

First we prove that (1) for $n-1 \leq i < 4(n-1)$, W increments *W.max_clock* by 1 between b_i and b_{i+1} and (2) for $n \leq i \leq 4n$, $W.max\_clock(b_i) \geq P.max\_clock(b_i)$ for all $P \in \mathcal{P}$.

Since Q works asap, W executes a single step between b_i and b_{i+1}. W does not increase $W.max\_clock$ by 1 only if W reads a value that is greater than $W.max\_clock + 1$. Assume towards contradiction that W reads from $P.clock$ (for some processor P) a value that is greater than $W.max\_clock + 1$. Recall that for any processor it holds that $max\_clock \geq clock$. By the definition of W, following b_n and before b_{n+1}, P reads a value of at least $W.max\_clock + 2$ from some $clock$ variable.

Let P' be the first processor that increases $P'.clock$ to be at least $W.maxclock + 2$. Since every processor executes at most one step between b_{n-1} and b_n and since by the definition of W, P' increases $P'.clock$ by more than 1, P' reads a value that is greater than or equal to $W.max\_clock + 2$ from another processor, which contradicts the definition of P'. The above implies that W increments $W.max\_clock$ by 1 and that no clock value is greater than $W.max\_clock(b_{n-1}) + 1$ in b_n. Thus, assertions (1) and (2) hold. Using the same argument we proof that assertions (1) and (2) hold starting with b_n and reaching b_{n+1}, and so on and so forth up to the end of E''_q.

We proved that W has the maximal $W.max\_clock$ in any b_i such that $n - 1 \leq i \leq 4(n - 1)$. In particular, if $W = Q$ we have shown that for all $P \in \mathcal{P}$, $Q.max\_clock(c_{3n-3}) \geq P.max\_clock(c_{3n-3})$. Next we show that this also holds when $W \neq Q$.

During the second $n - 1$ pulses, W assigns its clock to be the value of $W.max\_clock$. Thus $W.max\_clock(b_{2n-2}) = W.clock(b_{2n-2}) = W.max\_clock(b_{n-1}) + n - 1$. Moreover, following that assignment and throughout $\hat{E}''_q$, in every configuration b_i that immediately precedes a step of Q, $W.clock(b_i) = g\_clock(b_i)$ (g_clock is defined the same as for the in-phase system.) Before the third $n - 1$ pulses elapse, Q reads the value of $W.clock$. Thus, in the configuration c_{3n-3} of $\hat{E}''_q$ it holds that $Q.max\_clock(c_{3n-3}) = W.max\_clock(c_{3n-3}) = W.clock(c_{3n-3}) = W.max\_clock(c_{n-1}) + 2n - 2$ and for every processor $P \in \mathcal{P}$, $Q.max\_clock(c_{3n-3}) \geq P.max\_clock(c_{3n-3})$.

Thus, in order to show that $Q.max\_clock(\hat{c}) = g\_max\_clock(\hat{c})$ we only have to show that for every processor $R \notin \mathcal{P}$, $Q.max\_clock(\hat{c}) \geq R.max\_clock(\hat{c})$. Assume toward contradiction that $R.max\_clock(\hat{c}) > Q.max\_clock(\hat{c})$ for some processor $R \notin \mathcal{P}$. Since $Q.max\_clock(\hat{c}) = W.clock(\hat{c}) = g\_clock(\hat{c})$, then $R.clock(\hat{c}) < R.max\_clock(\hat{c})$. $R.max\_clock(\hat{c})$ may be greater than $R.clock(\hat{c})$ only due to a value R reads from a processor clock, say $P.clock$, during a step r that is one of the last $n - 2$ steps of R before $\hat{c}$. (Recall that during at most $n - 1$ successive steps of R there is a step in which $R.clock = R.max\_clock$.)

Consider two cases: (a) r occurred before c_{2n-2} or (b) r occurred after c_{2n-2}. In case (a) R does not execute at least $2n - 2 - (n - 2)$ steps (otherwise, R assigns $R.clock = R.max\_clock$ after r which contradicts the definition of r). Let c_r be the configuration that immediately follows r. Then $R.max\_clock(c_r) \leq g\_clock(c_r)$. As mentioned above R may increase the value read by at most $n - 2$ before R assigns $R.clock = R.max\_clock$. Thus, $R.max\_clock(\hat{c}) \leq R.max\_clock(c_r) + n - 2$. Moreover, since during any of the last $2n - 2$ steps W increments its clock by 1 (and hence increases g_clock in any configuration that immediately follows a step of Q) then $g\_clock(c_r) \leq g\_clock(\hat{c}) - 2n - 2$. Hence, $R.max\_clock(\hat{c}) \leq R.max\_clock(c_r) +$

$n-2 \leq g\_clock(c_r)+n-2 \leq g\_clock(\hat{c})-2n-2+n-2 < g\_clock(\hat{c})$. In case (b) R assigns $R.max\_clock$ to be a value such that for any $2n < i \leq 4n$ $R.max\_clock(b_i) \leq W.clock(b_i)$. Since $Q.max\_clock(\hat{c}) \geq W.clock(\hat{c})$ the contradiction is completed. ■

Lemma 5.4 *In every configuration c' in E_q after $\hat{c}$ that immediately follows a step of Q, $Q.max\_clock(c') = g\_clock(c') = g\_max\_clock(c')$.*

Proof: We prove the lemma by induction on the number of steps executed by Q following $\hat{c}$ during E'_q. The basis is by Lemma 5.3. Let q_i and q_{i+1} be two such successive steps of Q. We claim that between q_i and q_{i+1} a processor P may increase $P.clock$ and $P.max\_clock$, during a step p, to be at most the value of $Q.clock+1$. P may read at most the value $Q.clock+1$ and assign $P.max\_clock := Q.clock+1$. The proof is completed since during q_{i+1} Q increments both $Q.max\_clock$ and $Q.clock$ by 1. ■

Theorem 5.5 *The above algorithm is wait-free clock synchronization algorithm with convergence time $k = 24n^2+n$.*

Proof: By Lemma 5.4, Q has the maximal clock value after each of Q's step that follows $\hat{c}$. Thus the only change ever made to $Q.clock$ is to increment it by 1, as long as Q continues to work. This implies the adjustment requirement. If there is any other processor P that has been working long enough, then it too has the maximal clock value, which is equal to either $Q.clock$ or $Q.clock-1$ in any configuration that follows Q's step. Thus the adjustment requirement holds too. ■

6 Concluding Remarks

We have defined the new class of *wait-free clock synchronization* algorithms. Algorithms in this class may tolerate *napping* faults that cause a processor to repeatedly stop operation and then resume operation without necessarily recognizing that a failure occurred. Wait-free clock synchronization algorithms can tolerate up to $n-1$ faulty processors. The viability of this definition was demonstrated by presenting three wait-free clock synchronization algorithms, one of which is also self-stabilizing.

In a single-site multiprocessor system it may be possible to achieve tighter clock synchronization than in a multi-site distributed system. In fact, when a common clock pulse is assumed our algorithms cause the clocks to be perfectly synchronized. Such strong synchronization might be needed in systems where simultaneous actions are important, e.g., the firing squad synchronization problem (cf. [BL87, CDDS89]). When there is no common pulse we have to consider the skew between the individual pulses. Digital clocks that are incremented at different real time cannot be totally synchronized — they are doomed to have at least one unit difference.

Many interesting open questions remain.

A straightforward lower bound on k for any wait-free clock synchronization algorithm is $\Omega(n)$. There is a simple randomized self-stabilizing algorithm with expected $k = O(n)$. In the in-phase version of that randomized algorithm, during every step P chooses randomly the neighbor Q from which to read and assigns its clock to the maximum of $P.clock + 1$ and $Q.clock + 1$. An interesting open question is whether a deterministic protocol with $k = O(n)$ exists.

To complement work on lower bounds, finding improved upper bounds on k for clock synchronization would be helpful. It may be that the requirement for self-stabilization affects the achievable bounds. Preliminary follow-up work in this direction has been done, describing an $O(n^2)$ self-stabilizing algorithm [PT94].

In a very strong model of communication, in which at each global clock pulse every (non-faulty) processor first reads all n shared variables and then writes a new value to its own shared variable, self-stabilizing wait-free clock synchronization can be achieved with $k = \Theta(1)$ even when the clock values are bounded (i.e., the clock is a bounded counter, see the appendix for this algorithm.) It is currently unknown whether there exists an algorithm for bounded clocks in either our in-phase and out-of-phase systems (of Section 2).

More general future work in this area includes devising algorithms for other tasks (e.g., agreement, renaming) using the same model of faults and analogous requirements, and investigating lower bounds on k for each of them.

Acknowledgments: We thank Mohamed Gouda and Steve Liu for helpful discussions. Many thanks to the anonymous referees for a careful reading of a preliminary version of this paper and for their helpful suggestions.

References

[ADG91] A. Arora, S. Dolev, and M. Gouda, "Maintaining Digital Clocks in Step," *Parallel Processing Letters*, Vol. 1, No. 1, 1991, pp. 11-18.

[AKY90] Y. Afek, S. Kutten and M. Yung, "Memory-Efficient Self Stabilization on General Networks," *Proceedings of the 4th International Workshop on Distributed Algorithms*, Bari, Italy, September 1990, LNCS 486 Springer-Verlag, pp. 15-28.

[ALS90] H. Attiya, N. A. Lynch and N. Shavit, "Are Wait-Free Algorithms Fast?," *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science,* October 1990, pp. 55-64.

[AV91] B. Awerbuch and G. Varghese. "Distributed Program Checking: a Paradigm for Building Self-Stabilizing Distributed Protocols," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, October 1991, pp. 258-267.

[BL87] J. E. Burns and N. A. Lynch, "The Byzantine Firing Squad Problem," *Advances in Computing Research: Parallel and Distributed Computing*, Vol. 4, 1987, pp. 147-161.

[CDDS89] B. A. Coan, D. Dolev, C. Dwork and L. Stockmeyer, "The Distributed Firing Squad Problem," *SIAM Journal on Computing*, Vol. 18, No. 5, 1989, pp. 990-1012.

[Di74] E. W. Dijkstra, "Self Stabilizing Systems in Spite of Distributed Control," *Communication of the ACM*, Vol. 17, 1974, pp. 643-644.

[DHS86] D. Dolev, J. Y. Halpern, and H. R. Strong, "On the Possibility and Impossibility of Achieving Clock Synchronization," *Journal of Computer and System Sciences*, Vol. 32, No. 2, 1986, pp. 230-250.

[DIM90] S. Dolev, A. Israeli, and S. Moran, "Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity," *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, August 1990, pp. 103-117.

[DIM91] S. Dolev, A. Israeli, and S. Moran, "Resource Bounds for Self Stabilizing Message Driven Protocols," *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, August 1991, pp. 281-293.

[DW93] S. Dolev and J. Welch, "Wait-Free Clock Synchronization," *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, August 1993, pp. 97-108.

[GH90] M. G. Gouda and T. Herman. "Stabilizing Unison," *Information Processing Letters*, Vol. 35, 1990, pp. 171-175.

[GP93] A. Gopal, and K. J. Perry, "Unifying Self-Stabilization and Fault-Tolerance", *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, August 1993, pp. 195-206.

[HSSD84] J. Halpern, B. Simons, R. Strong, and D. Dolev, "Fault-Tolerant Clock Synchronization", *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing,* August 1984, pp. 89-102.

[Hw93] K. Hwang, *Advanced Computer Architecture, Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., 1993.

[KP+92] Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan, "Efficient Program Transformations for Resilient Parallel Computation via Randomization," *Proceedings of the 24th ACM Symposium on Theory of Computing,* 1992, pp. 306-317.

[KS91] P. C. Kanellakis and A. A. Shvartsman, "Efficient Parallel Algorithms on Restartable Fail-Stop Processors," *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing,* 1991, pp. 23-36.

[KR90] R. Karp and V. Ramachandran. "Parallel algorithms for shared memory machines," *Handbook of Theoretical Computer Science,* J. van Leeuwen, ed., Elsevier Science Publishers B.V. (also MIT Press), 1990, pp. 869-941.

[La86a] L. Lamport, "The Mutual Exclusion Problem: Part II - Statement and Solutions," *JACM*, Vol. 33, 1986, pp. 327-348.

[La86b] L. Lamport, "On Interprocess Communication," *Distributed Computing,* Vol. 1, No. 1, 1986, pp. 86-101.

[Le92] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, Inc., 1992.

[LM85] L. Lamport and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *Journal of the ACM*, Vol. 32, No. 1, 1985, pp. 1-36.

[Ma83] K. Marzullo, *Loosely-Coupled Distributed Services: A Distributed Time Service,* Ph.D. Thesis, Stanford University, 1983.

[MC80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[Mo64] E. F. Moore, "The Firing Squad Synchronization Problem," *Sequential Machines*, ed. E. F. Moore (Addison-Wesley), 1964.

[MS85] S. Mahaney and F. Schneider, "Inexact Agreement: Accuracy, Precision and Graceful Degradation," *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing,* August 1985, pp. 237-249.

[PT94] M. Papatriantafilou and P. Tsigas, "Self-Stabilizing Wait-Free Clock Synchronization," Technical Report CS-R9421 Centrum Voor Wiskunde en Informatica, 1994.

[ST87] T. K. Srikanth and S. Toueg, "Optimal Clock Synchronization," *Journal of the ACM*, Vol. 34, No. 3, 1987, pp. 626-645.

[Ul84] J. D. Ullman, *Computational Aspects of VLSI,* Computer Science Press, 1984.

[WL88] J. L. Welch and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation*, Vol. 77, No. 1, 1988, pp. 1-36.

A Bounded Algorithm for In-Phase System with Global Read

Recall the simple in-phase algorithm presented at the beginning of Section 3.1 assuming a processor can read the entire shared memory at each step. If the algorithm is modified so that the clock is incremented by one modulo some maximal clock value (instead of using unbounded values), then the algorithm does not fulfill the adjustment requirement: if a processor with the maximal clock value is napping forever then it causes every other processor to repeatedly adjust its clock.

We now present an in-phase algorithm that uses a bounded amount of memory, assuming that a processor can read all the shared variables at each step. In particular, it is designed for the case of bounded clocks, meaning that the clock variable can only take on integer values between 0 and $M-1$, for some positive integer M. Note that in this case, the adjustment condition is modified so that the arithmetic is modulo M.

In addition to the *clock* variable, each processor P has, for every other processor Q, a *count* variable $P.count[Q]$. Each *count* variable may hold one of three values: 0, 1, or 2. For every two processors Q and R, we say that Q is *behind* R if $Q.count[R]+1 \pmod 3 = R.count[Q]$. Roughly speaking, the *count* variable ensures that a napping processor will be ignored since it will be behind every non-napping processor. Every non-napping processor tries to make each neighbor be behind itself and succeeds only if this neighbor is napping.

At each step P reads all the *count* and *clock* variables in the system and executes the following:

1. Let $\mathcal{R}$ be the set of processors that are not behind any other processor.
2. If $\mathcal{R}$ is not empty then $P.clock := R.clock + 1 \pmod M$, where R is the processor with the maximal clock in $\mathcal{R}$.
3. For every processor Q, if Q is not behind P then $P.count[Q] := P.count[Q]+1 \pmod 3$.

Theorem A.1 *The above algorithm is a self-stabilizing wait-free clock synchronization algorithm with convergence time $k = 2$.*

Proof: Fix an execution E. First note that all processors that take a step at the same pulse see the same view and compute the same $\mathcal{R}$.

Suppose processor X takes a step at pulse $T-1$. Then X is not behind any processor in configuration $T-1$. Thus all processors that take a step at pulse T compute the same value for $\mathcal{R}$, which includes X, and set their clocks using the maximum clock value among the processors in $\mathcal{R}$.

Agreement: Suppose $work(P,E,T) \geq 2$ and $work(Q,E,T) \geq 2$. By the above argument, P and Q set their clock the same way during pulse T and $P.clock(T) = Q.clock(T)$.

Adjustment: Suppose $work(P,E,T+1) \geq 3$. We show that $P.clock(T+1) = P.clock(T)+1 \pmod{M}$. By the above argument, P is in $\mathcal{R}$ in configuration T. So P sets its clock to $R.clock+1 \pmod{M}$ during pulse $T+1$, where R is a processor in $\mathcal{R}$ with the maximum clock value. The proof is clear when R is P. Assume that $R \neq P$. Since $work(P,E,T+1) \geq 3$ it holds that $P \in \mathcal{R}$ in configuration $T-1$. In particular, $\mathcal{R}$ is not empty in configuration $T-1$. By the fact that R is not behind P in configuration T it holds that R took a step at pulse T (as P did). Thus, during pulse T both P and R set their clock the same way and $P.clock(T) = R.clock(T)$.

The above arguments assume an arbitrary starting configuration for the execution, with any combination of *count* and *clock* values. Thus the algorithm is also self-stabilizing. ■